An Architectural Style to Integrate Components and Aspects

Miguel A. Pérez, Amparo Navasa, Juan M. Murillo
Area de Lenguajes y Sistemas Informáticos
Departamento de Informática. Extremadura Univ.
{toledano, amparonm, juanmamu}@unex.es

In the last few years Component Based Development (CBD) and Aspect Oriented Programming (AOP) are gaining more and more relevance. Whilst CBD has been shown as a good mean to reuse designs and to build complex systems by the plug-and-play mechanisms, AOP makes code more reusable and complex systems more comprehensible. However, little efforts have been made to integrate both paradigms and consequently, developers are condemned to don’t get all the advantages offered by the two paradigms simultaneously. In this paper a proposal to integrate AOP and CBD is presented. The work introduce the concept of Aspect Component and it is split in two different parts: first, an architectural style to support both, functional and aspect components is proposed and second, an approach to document, search and find aspect components in repositories is presented.

1 Introduction

In the last few years, AOP and CBD have become more relevant, and have showed their use in developing complex systems. Nevertheless, both paradigms have evolved in separate ways:

1. AOP has been shown as a good mean to develop complex systems. This paradigm allows separating the aspect code that contaminates the functional code of applications. Some of the main benefits obtained are to have both, functional and aspects code, more reusable and make the systems more comprehensible. As a consequence the software quality is improved.

2. On the other hand CBD is shown as a powerful paradigm that favours the designs’ reuse, using the plug and play mechanism to build software systems. The systems that are built in this way evolve in a simple fashion and are easily maintained. However, in this paradigm, the aspect code is crosscutting the functional code in components.

Although AOP and CBD are conceived to develop complex Software System, this paradigms can not be used both together. The reason is that the actual CBD models do not support aspect separation. This means the developers must trade off one paradigm for the other, and means that they are unable to obtain benefits from both paradigms simultaneously. Nevertheless, intuitively, it seems possible that both paradigms can be joined together successfully [ScAs98]: aspects could be separated from components to be modelled as a special kind of component. This would allow giving the benefits of components to aspects and the benefits of separation of concerns to components.

In this paper, a proposal to combine AOP and CBD is presented. The primary objective is to construct systems based on components that separately treat several

1 This work has been supported by CICYT, project TIC 99-1083-C2 –O2.
(functional and non-functional) concerns. To do this, the concept of *aspect component* is introduced and defined in the following way:

*An “aspect component” is a component that has a code associated with a concern (and only one), functional or non-functional.*

The contribution of the paper is two-fold: on one hand, it defines an architectural style in the sense [GaSh96] for the construction of component oriented systems, considering non-functional aspects. On the other hand, the work assumes the existence of repositories of aspect components. Retrieving the information from them presents different problems than those already know for the functional component repositories. For these reason it is introduced a model for the documentation, selection and retrieval of aspect components. The model is based on the analysis of the information of interoperability needed for this special kind of components.

### 2. Proposed model

The first contribution of this paper consists of defining an architectural style for the definition of components oriented complex System. It considers the non-functional aspects that are applied to functional components. In this model, we consider the following definition of a component give by Krunchen in the CBSE workshop in 1998 [Kr98]:

*A component is a non-trivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of a well-defined architecture. A component conforms to and provides the physical realisation of a set of interfaces.*

The software architecture is based on the layered architectural style proposed by Garlan and Shaw [ShGa96]. In the architectural style proposed there is one level for each considered concern (figure1). Each level controls the actions performed in the inferior level according to the requirements of the concern that is implementing. Also, each level is intended to support the plug in of the adequate aspect components.

![Figure 1. General architecture](image-url)
The proposed model is based on the following principles:

1. The components that form part of the system have a generic structure. Never they will be considered built by or for the model. Thus, one can suppose that they can be found in public repositories.

2. Each component has an interface to present its own services.

3. Each component needs to be associated with information of interoperability in order to locate them in the repositories and know if their requirements coincide with the sought requirements.

Figure 1 represents the proposal graphically. It indicates the order in which the aspects should be considered. Note that, whilst the second and third levels are only applied to one component, next levels are applied to several ones. The reason is that co-ordination and distribution concerns relate several components.

Levels in proposed architectural style are:

**Level 1**: The one called “functional components level” is the lowest. It contains the functional components. Each one has one or more interfaces to define its interaction with the others components in the system.

**Level 2**: Concurrency Level. In this level, the concurrency control for components in level 1 is made. Components in this level include constraints specifying how the actions performed by the functional components can be executed concurrently.

**Level 3**: Synchronisation level. This level will decide when, the actions performed by the functional component can be executed. This is made according to the constraints specified by the aspect components plugged in at this level.

**Level 4**: Distribution level. Using a platform (CORBA, RMI,...) it is possible to access remote components. Platforms manage the components location and their communication. This level is defined over components in previous levels. It defines the distribution policy of components in the system. The model presented here doesn’t wish acquire obligations with any platform, but considers the possibility to use several at one time.

**Level 5**: The co-ordination level implements the dependencies among the actions performed in the whole system. Components at this level specify the co-ordination constraints to apply over the system components (distributed or not).

Each level controls the action performed in the inferior level through a reflective schema: When a service is required to a functional component the requirement is intercepted and processed by the superior levels.

This architecture allows building systems based on functional and non-functional components. However, it is necessary note that components will reside in repositories and that they should have a good interoperability specification for making correct matches when searching them. Next section study this problem.
3 Component selection

The second part of the work presented here consists of searching for and selecting aspects components from repositories. If it is well known the difficulty to retrieve functional components in repositories [HeVaTr00, VaHeTr99], the retrieval of the aspect components is even more complicated. The reasons are the following:

- Aspect components can be developed for different models of separation of concerns [Berg94, Kic96, Lie96, Cza98, Osh96]. Each one of these models is different from the others, which complicates the recuperation. It is needed model independent documentation of interoperability, that is: the documentation of interoperability must be valid for all the designed components for the same aspect (independently of the model that the designer choice to achieve the separation).
- The architectural style proposed collects various aspects, and each aspect needs different interoperability documentation. For example, the semantic description of a component that specifies the co-ordination aspect is different from one specifying the distribution aspect. This is due to the fact that the documentation requirements change when describing each behaviour.
- While the interface of the functional components serves to show the services that this particular component offers, in the aspect components the interface describes the requirements required for the components to which they will be applied.
- Besides, to select a functional component from a repository we need key words, names of components, domains, granularity,... Nevertheless, to select aspect components, apart from those mentioned above, we need to access components in relation to their behaviour, and this is difficult to describe.

Whilst system’s construction using functional components [ScAs98] has already been handled by methodologies like Catalysis [SoWi00], composition using aspect components lack methodologies, and there are not any public aspect repositories, nor tools to select them.

Our work considers that there exists aspect repositories from which to select the desired components. We have found that to select aspect components, it is needed documentation that includes the following:

1. The documentation of interoperability of the component should contain syntactic information that describes the aspect component’s interface. This description could be one that explains to the other components how they should use the aspect component.
2. Semantic information that explains the aspect component’s behaviour. This information is more difficult to obtain, and requires the use of specific tools, since the needs of documentation change for every aspect. The objective is to generate information about what the component does.
3. General information, about the model of the component, the name of the component, the most frequently used domain, the size,... This information can be completed with information about the transitions of the component’s state to facilitate its comprehension. The idea is to obtain complementary information, which permits a more complete knowledge of the desired component and facilitates its interoperability in different application domains.
The information in point 1 is similar to the syntactic information necessary to use any functional component. But now, this information does not exhibit the services, but rather the requirements, which the components that are used should comply. The information obtained in point 3 is not different from that which a functional component could have. However, the semantic information in point 2 presents new challenges.

To describe semantic information, one needs a tool to describe the behaviour of the different aspect components, permit the selection of components from a repository and compare the behaviour of different components in order to choose them correctly.

Our research began with a study of the co-ordination components. Thus, to document the components [Kr98] that collect the co-ordination aspect, we use diagrams similar to sequence diagrams proposed in UML. However, we have amplified its syntax to cover the needs of the aspect of co-ordination which do not collect the diagrams of sequence [In00, HaPo98], such as representing the notification of event or differentiating control method of the notification of events.

Once we have obtained the diagram(s) that describe the function and interaction with other components, the tool constructs (based on it) the automaton that describes its behaviour. Finally, this automaton is optimised. The obtained automata will be used to search, compare, and select elements from repositories.

Our tool is presented in a graphic environment to collect, in a simple fashion, the entire user’s needs: the sequence of events that the component might receive, the type, the form of notification… From these needs automata are built that are compared with those represented in the repository.

The following steps are for the creation of new tools that permit us to work with other different aspects. The final goal is to have an adequate environment to select any type of aspect components.

4 Conclusions.

In this paper, it has been presented a software architecture for building complex systems integrating concepts of AOP and CBD. Also, it has been presented a proposal for the documentation and selection of aspect components from repositories.

Future works that we have proposed are directed toward the formulisation of architecture and the creation of tools for the selection and retrieval of aspect components.

5 Bibliography


[SoWi00] Francis D’Souza, Cameron Wills, “Objects, Componentes and Frameworks with UML. The Catalysis Approach”